DOCUMENT RESUME

ED 234 759 IR 010 826

AUTHOR	Allen, M. F.
TITLE	Teaching BASIC. A Step by Step Guide.
INSTITUTION	People's Computer Co., Menlo Park, CA.
PUB DATE	83
NOTE	65p.
AVAILABLE FROM	People's Computer Co., PO Box E, Menlo Park, CA 94025 ($9.00, per copy plus $1.25, shipping and handling).
PUB TYPE	Guides - Classroom Use - Guides (For Teachers) (052) -- Audiovisual Materials (100)
EDRS PRICE	MF01/PC03 Plus Postage.
DESCRIPTORS	Adolescents; Adults; Computer Literacy; Computer Oriented Programs; *Computer Programs; Instructional Materials; Introductory Courses; *Microcomputers; Programing; *Programing Languages; Teaching Guides
IDENTIFIERS	*BASIC Programing Language; ComputerTown

ABSTRACT

This three-chapter guide provides simple explanations about BASIC programming for a teacher to use in a classroom situation, and suggests procedures for a "hands-on" course. Numerous examples are presented of the questions, problems, and level of understanding to expect from first-time, adult users (ages 13 and up). The course materials are designed to work for all microcomputers and cover the Introduction to BASIC Course (representing 3 2- to 3-hour classes), as presented in ComputerTown classes for the general public. The goal is for students to understand the commands as tools, to gain an overall picture of how a microcomputer works, and to have a positive experience with computers. Topics in the first lesson include hardware, software, memory, commands (PRINT, LIST, RUN, GOTO), editing, and the break key. The second lesson covers variables, counting to infinity, INPUT, IF-THEN, and programming syntax and theory, while the third introduces string variables, nested loops, and the STEP and FOR-NEXT commands. An appendix contains the following course handouts: (1) Introduction to BASIC Programming; (2) BASIC: Getting Turned On; (3) Big Bucks Banking Automated Checking Program; (4) Rules for Variable Names; (5) Number Guessing Game; and (6) A Word on Buying Software. (LMM)

****
* Reproductions supplied by EDRS are the best that can be made *
* from the original document. *
****

ED234759

U.S. DEPARTMENT OF EDUCATION
NATIONAL INSTITUTE OF EDUCATION
EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)

☙ This document has been reproduced as received from the person or organization originating it.

☐ Minor changes have been made to improve reproduction quality.

• Points of view or opinions stated in this document do not necessarily represent official NIE position or policy.

A STEP BY STEP GUIDE

TEACHING BASIC

by M. F. Allen

ComputerTown GUIDEBOOK

© 1983 People's Computer Company

IR010826

"PERMISSION TO REPRODUCE THIS MATERIAL HAS BEEN GRANTED BY
Fritzi Lareau
TO THE EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)."

A STEP-BY-STEP GUIDE TO TEACHING BASIC

TABLE OF CONTENTS

PAGE

INTRODUCTION:

What This Booklet Offers 1

About The Course And Who Should Take it 1

About Teaching This Course 2

About The Layout Of This Booklet 4

LESSON ONE:

First Things First: Why Are You Here? 5

Some Course Groundrules 6

Notes To The Instructor 7

Hardware 7

Software 8

Memory 8

Turning On 9

Communicating With The Computer 10

The First Command: PRINT 11

When To Hit Return 12

PRINT With A String 12

Direct Mode 13

Program: A Line Number & A Command 13

RUN, Output, LIST 14

Editing 15

PRINT With A Value 17

Starting A New Program: NEW 19

The GOTO Command 19

The Break Key 20

CONTINUED

TABLE OF CONTENTS

LESSON TWO:

Review 22

Variables 23

PRINT A Variable's Value 24

Counting To Infinity 25

Out Of Memory (Optional) 27

Involving The User: INPUT 27

Directions With INPUT 29

Rules For The Names Of Numeric Variables . . . 30

Big Bucks Banking Program 31

Programming: Syntax Vs. Theory 32

The IF-THEN Command 32

Modifying The Banking Program With IF-THEN . . 33

LESSON THREE:

String Variables 33

Rules For String Variable Names 39

Modifying The Banking Program, String Variables.42

Coordinating The Whole Program 42

When The User Does Not Follow Directions . . . 44

The FOR-NEXT Command: A Controlled Loop 45

Using STEP (Optional) 47

Nested Loops 48

Modifying The Banking Program With FOR-NEXT . . 50

The Last Word 51

CONTINUED

TABLE OF CONTENTS

APPENDIX OF HANDOUTS 53

Introduction To BASIC Programming

BASIC: Getting Turned On

Big Bucks Banking, Automated Checking Program

Rules For Variable Names

Number Guessing Game

A Word On Buying Software

INTRODUCTION

WHAT THIS BOOKLET OFFERS

After more than 300 students myself, and another 700 taught by the other ComputerTown teachers, I begin to suspect that there are some universal questions, points that consistently give more trouble than others, and concepts that we as experienced programmers and users have trouble explaining in simple terms. Thus this booklet.

This guide to teaching BASIC programming provides:

1. Clear, simple, non-mathematical explanations about BASIC programming for a teacher to use in a classroom situation;

2. A course designed to be taught entirely "hands-on";

3. Numerous examples of what questions, problems, and level of understanding to expect from first-time adult users (ages 13 and up);

4. A course and material designed to work for all microcomputers.

For more than a year, ComputerTown has been offering beginning computer classes to the general public, from childrens' courses, to adult courses (ages 13 and up) in Computer Comfort, Introduction to BASIC, and Intermediate BASIC.

This booklet covers the material of the Introduction to BASIC course. The three chapters here represent approximately six to ten hours of instruction. Each chapter is a two to three hour class, depending on the group of students, practice time, and the material you may expand upon or leave out.

ABOUT THE COURSE AND WHO SHOULD TAKE IT

Introduction to BASIC, and any ComputerTown course, is for people with no previous experience, who are curious, and want a hands-on course with computers. The math is no more rigorous than some addition and subtraction. No homework is required, although some is suggested. Ideas for

assignments are included.

ComputerTown often gets students who have previously taken an academic programming course that was in a lecture format. Students were expected to que up for many hours on their own time to work in the computer lab. We also get high school students who are not eligible for computer classes at their own school because they lack some math prerequisites. We have had students from ages 13 to 81 take this course, housewives, senior citizens, and businessmen.

The purpose of the course is not to produce programmers, although it is an excellent foundation for more advanced courses. Most people will never be programmers, but they are likely to be users of other people's programs. Students should understand the commands as tools. The overall picture is crucial: how a computer works; that logic is the key; that it does only one thing at a time; that it does not know anything the programmer does not tell it; that it cannot function with even a small error in the program. It is most important that students go away with a positive attitude, from a positve experience with computers.

ABOUT TEACHING THIS COURSE

Use the explanations here to spark your own ideas on how to most clearly introduce the material. Also take note of sections where I have pointed out the conceptual stumbling blocks that the ordinary student will have.

More importantly, remember the fundamental details you now take for granted. As an experienced user, you may have unforseen problems with this. Do not forget to explain what the cursor is, when to hit RETURN, what that means, etc. This has consistently been the greatest shock to first-time instructors: the myriad things they do or use unconsciously on the computer.

Overlooking detail can be one unexpected problem for the instructor. Another major problem lies in the opposite extreme: You must learn not to tell the class EVERYTHING you know. 1) There is not enough time in six to ten hours; 2) You will sacrifice most of the class to boredom. The tendancy is powerful, especially if it is some area you have a penchant for yourself. Explain the most basic use of a command. NEVER explain all of the exceptions. In general, don't explain any of them. You may say exceptions exist, or assign them for homework. Quicker students can be presented with extras to experiment with while you work on other students' problems.

Do not let the students lead you too far astray with their questions. Never hesitate to say "we will get to that

later," or "we won't cover that in this class." Do try to work their questions into the lecture, letting the question lead into the next command or explanation. If the progression of the lecture can come out of their questions, it reinforces the underlying logic of programming. For instance, just when you have explained how to edit a line, some one will ask "how do you get rid of a line altogether?"

The third big problem for instructors: unconscious use of jargon. Try explaining to a friend, who knows nothing about computers, the parts of the computer and how to turn it on, or the use of the PRINT command with a string. Have the friend write down every word you did not explain which he did not understand. You could be amazed at the list.

If a term is fairly straight forward english, e.g. STATEMENT, or VIDEO SCREEN, then I do not pointedly define it in lecture. However, beware: we the instructors are not the best judges of what is "straight forward english" and what is not. Use as much real, descriptive english as possible. Keep new vocabulary to a minimum.

Of course it is "wrong" to refer to the computer as an animate entity. In this booklet I have often used "thinking" to refer to how the computer is processing information. I do this to avoid a more technical, lengthy explanation. If you also use "think" in this manner, be sure to explain at the first class that computers do not really "think."

Outside lab time and homework make an enormous difference in the amount of material students can retain and how fast they can move. Imagine taking a spanish course and never memorizing any vocabulary. This is analogous to a course in BASIC without homework. Many situations involving adults, however, do not afford the luxury of lab time and homework. It is important then to emphasize the ideas, the logic. Flowcharting is not necessary in this particular course if you emphasize logical progression and stepwise refinement.

Quiz the group orally to reinforce ideas. Leave sentences unfinished and prompt them to respond.

Make the lecture animated, and interactive. Be friendly. Generally in the first class you are dealing with a group of anxiety-stricken people. Initially the group will be reticent. Force some responses (but don't frighten). There is a talker in every group. Take advantage of that person to get the others going. People learn much better when they are relaxed. Work at being sensitive to them and allaying their fears.

Pace the room constantly when you are not writing or demonstrating at the board, to see if they are

understanding or having a particular difficulty. Especially in the beginning, this observation will clue you in on points you need to cover in more detail or may have overlooked.

The materials I use in the class include handouts and a blackboard. The specific equipment I used in ComputerTown classes included: Atari 400/800, Apple II and II PLUS, Commodore's PET and CBM, and the Radio Shack TRS-80, model I. Some machines had tape players, some had disk drives, some had neither. Students should bring a pen to take notes. They can write on their handouts. They do not have to buy a book or cassette or diskette for this course. The students prefer to have one machine per person. However, when two students share they tend to learn more. When students share, make sure they do take turns at the keyboard.

Remember: the key objective is for the students to have a positive, enjoyable learning experience.

ABOUT THE LAYOUT OF THIS BOOKLET

There are three chapters which follow this introduction. Each chapter is a two to three hour class "lecture."

After the initial comments in chapter one, the text becomes a lecture format. Thereafter, information to you the teacher is contained in parenthesis surrounded by asterisks, e.g.:

(NOTE:)

When explaining a command, generally I have tried to introduce the command in theory, and give the syntax; let the students run the command; then cover a step-by-step execution of the command at the blackboard; and end with a recap of the precise syntax required.

In the strings, I generally use my own name because it is short, and helps the students remember what my name is. It would probably be a good idea to change the strings to your name.

The appendix at the end of the booklet contains the handouts mentioned in the lecture text.

LESSON ONE

FIRST THINGS FIRST: WHY ARE YOU HERE?

When the class is seated and quiet, have each student briefly state whether he has had any previous computer experience and why he is taking the class. This allows you to A) gauge how long your opening remarks about computers need be; and B) correct any misconceptions about what "Introduction to BASIC Programming" really is.

Most people enroll in this course for the right reasons: they want to learn the jargon and what it means to program a computer. Responses usually run like these: "I want to know what I'm reading about in the newspaper," "I want to know what my kids are doing," or "My business will have to get one so I need to start learning about computers."

A few students, however, come looking for something else. These are the most frequent erroneous notions I have encountered:

1. "This is a course in the basics of computing."

Here you should explain that BASIC is a specific LANGUAGE. It stands for "Beginner's All-purpose Symbolic Instruction Code," and is used to write programs for the computer. Students may have heard of other programming languages such as Pascal, COBOL (common business oriented language), or FORTRAN (formula translation). There are many programming languages, one of which is BASIC.

2. "I want to program our new office microcomputer to do the inventory."

To do this type of sophisticated programming would require, for most people, a minimum of two years of rigorous training. However, they should not despair. What they learn in this course will be helpful no matter how they use a computer. They may also find that using some one else's program (a software package) is a much better solution to their programming needs.

3. "I'm going to make a career change, and want to try programming."

Again, the ability to write sophisticated programs takes a great deal more training than students will get in this six hour course. But this course is a good place to start. At the end of it, the student may find that programming does not suit him. Or he may find programming suits him so well that he will do extensive reading, or go off to a junior college, or to a university, to learn more.

4. "I'm going to buy a computer for my office and I want to know what kind I should buy."

Learning how to make an intelligent choice between today's microcomputers is an entire course in itself, and "Introduction to BASIC" is not it. In fact the "Computer Comfort" course is a much better overview of the machines themselves.

Let the student know that a computer is chosen on the basis of its software. See the Appendix for "A WORD ON BUYING SOFTWARE." Give the student a copy of that page, advice after class, or explain that the information in this course will help, but will not make the decision for him. Do not spend much class time on this since it is not, afterall, the purpose of the course.

The dominant tone of the responses will probably be apologetic, for not having had previous experience, with a good dose of anxiety about computers. Be sensitive to this. Positively state that this is a course designed for beginners with no previous experience. Tactfully correct erroneous notions, like those above. It is absolutely necessary for them to be comfortable from the start.

Do tell the students what to expect: This is a focused course, in which they will be learning a particular programming language, BASIC. They should get an idea of what it is like to program a computer. They will type in short programs, and modify others, which demonstrate the elements of BASIC programming. (For an OVERVIEW of the microcomputer field, and what computers can do in general, students should take Computer Comfort.)

SOME COURSE GROUNDRULES

No homework is required in this course, although it can be incorporated. Do encourage the students to take notes during class and to review just before the next class.

Most adults have anxieties about forging into the unknown. Reassure them that nothing terrible can happen to the computer. It is virtually impossible to break these machines, short of throwing one on the floor. Nothing the students can type in will hurt the computer. Encourage them to experiment, especially with those keys not found on a typewriter.

Tell them that they must ALWAYS ask questions when they do not understand something, anything. As instructor you should never continue covering new material when facing a sea of blank stares. Pause at regular intervals and ask sincerely "Are there any questions?" Be suspicious if there are not.

Next distribute hand-outs for the first class and have each student sit down in front of a computer. If you are working with several different models in one room, take advantage of this by pointing out that different manufacturers put different packages around essentially the same thing. All the machines speak BASIC, but they can look quite different. Tell them to look at the front of the machine to determine the manufacturer.

NOTES TO THE INSTRUCTOR

(NOTE: From this point on in the booklet, the text is largely a transcribed lecture, except for any information addressed directly to you the instructor. Notes to the instructor will appear inside of asterisks and parenthesis.)

A WORKING VOCABULARY: PARTS OF THE MACHINE

HARDWARE

(NOTE: As you explain the hardware, walk around the room and put your hands on the parts of the computer. Students should have their machines turned on 15-20 minutes after walking into the room. They are most eager to "see it work," so keep up the pace through HARDWARE and SOFTWARE.)

Everything you can put your hands on is the HARDWARE. The biggest piece of hardware is usually the VIDEO SCREEN, or MONITOR. If your video screen is a TV set, notice that it is connected to the computer by a cable and SWITCHBOX, set to "game."

The next big piece of hardware is the KEYBOARD. Look at the keys, especially around the edges of the keyboard, and notice the ones NOT found on a typewriter. Under the keyboard is the actual computer. The computer is not in the monitor.

The smaller boxes with little doors on the front are the DISK DRIVES. Some students may have TAPE PLAYERS instead of disk drives. Their purpose will be explained shortly.

The Atari and Radio Shack computers have another small black box called a POWER SUPPLY. The Commodore and Apple

have a power supply too, but the manufacturer put it inside the body of the computer. The power supply transforms the alternating current coming from the wall outlet, into the direct current needed by all microcomputers. So if it is a separate piece of hardware, the power supply should always be plugged-in to both the wall and the computer.

Questions?

SOFTWARE

SOFTWARE makes use of the hardware so that the computer can perform useful tasks, such as programming, calculating difficult formulas, word processing, financial planning, etc. Software is a collection of instructions, or a program, which tells the computer to do something.

Some information is stored permanently in the computer. There are also two methods for you to put information into the computer: by typing it in, or by loading it in through the tape player or disk drive. In a computer store, you can buy software programs on a cassette tape or diskette. These programs are loaded into the computer through the disk drive or tape player.

In this class you will be writing programs in BASIC, and typing them into the computer via the keyboard.

MEMORY

Computers cannot hold an infinite number of instructions. The amount of information they can store is determined by the size of their MEMORY. The memory is measured in units called BYTES. A byte is approximately the amount of space taken up within the computer's memory by any single key you hit on the keyboard. Any character, letter, punctuation mark, number, the space bar, takes up about one byte of memory. We refer to bytes in thousands or K: so an 8K machine can store about eight thousand characters you type in through the keyboard.

There are two memory areas of the computer we are concerned with in this class: ROM and RAM. ROM is "Read Only Memory." ROM contains software put in the computer by the manufacturer, and there is nothing you can do to change it. Regardless of how many times you turn the computer on and off, that information stays on the ROM. The computer's understanding of the language BASIC is stored on the ROM in these machines.

*(NOTE: If you are using Apple IIs, rather than the II Plus, you might explain here, or soon after, that Apple developed two versions of BASIC: Integer BASIC and Applesoft BASIC. Applesoft BASIC is the more powerful and

more universal BASIC which is used in this class. Apple IIs have Integer BASIC in their ROM. Applesoft BASIC must be loaded into the computer, through the disk drive, onto the RAM. This situation affords a good opportunity to demonstrate the difference between the ROM and the RAM.)*

RAM is "Random Access Memory," and that is your workspace. Everything you type in through the keyboard, or load in via cassette or disk drive goes onto the RAM. The RAM holds information only as long as there is electricity coming into the computer. As soon as the computer is turned off, the RAM is wiped clean, dumped. So if you sit down and type in a program of many hundreds of lines, then turn off the computer, the program is gone for good. This is why we have diskettes and cassette tapes.

It is possible to type a very long program into the computer, on the RAM, and then save a copy of the program onto a diskette or cassette. So when you turn off the computer and the RAM is dumped, you still have a copy of the program. The next day when you sit down to use that program or to add to it, you need not type the whole thing onto the RAM again. You insert your diskette (or tape) into the disk drive (or tape player) and command the computer to load a copy of the program from the diskette onto the RAM. Typing in the program can take hours. But the computer loads or saves a copy of the program in a few seconds.

(NOTE: If time permits at the end of the second class, let students save the banking program for the next class. If you plan it as a definite part of the course, tell the students so now. In this course, for the most part, they will concentrate on typing programs into the computer rather using peripheral storage devices.)

Questions?

*(NOTE: If you have disk drives, you might boot a DOS now, show a CATALOG of programs on a single diskette, LOAD a program, then RUN a program to demonstrate the peripheral storage. Do not try to explain exactly what you are doing, just the abstract side of loading/saving, and the temporary nature of the RAM.)

TURNING ON

*(NOTE: From this point on, remember the cardinal rule: as instructor, you should NEVER put your hands on a student's computer. The temptation is great. But adults will never become confident if you continually work hocuspocus across their keyboards or in those mysterious regions beyond where the plugs are attached. Always prod them with suggestions or leading questions, but try not to do anything for them. Where I have posed

questions, encourage the students to answer. Once someone does answer with at least the right idea, then repeat the answer, articulating with computer vocabulary.)*

I want everyone to turn on their systems. Remember: If monitor and computer are two separate pieces of hardware, you will have to turn on two things.

(NOTE: If there are many types of computers in the room, make up a sheet similar to the TURNING ON handout in the appendix of this booklet. Talk to them as they search for switches and plugs. WARNING: If you are using the Commodore PET computer, watch that students do not pull out the fuse as they search for the rocker switch.)

The switch for the computer may not be in an obvious place. Feel around the edges and the back for it. Why would the computer designers put the switch in a spot that is not easily accessible, some place where you're not likely to turn off the computer accidentally..? When you turn off the computer, the RAM gets dumped, all your work gets wiped out. This can be a great disaster if you have not saved a long program or data. Thus the switch must be located out of range of straying fingers and elbows.

Some of these computers give you information as soon as they are turned on. For instance, the Atari says "READY." The PET identifies itself and tells you how many bytes you have on your workspace, the RAM. Where is the information, there right when you turn on the computer, coming from..? The ROM, information put there by the manufacturer. Apple Computer Company simply decided not to tell you anything when you turn on the Apple II Plus.

Questions?

COMMUNICATING WITH THE COMPUTER

Now type in your name or hello or anything you want. That square which moves, and in some cases blinks, is called the CURSOR. The CURSOR indicates where you are, as far as the computer is concerned, at any time.

When you finish hit the RETURN key, ENTER on the Radio Shack computer, to enter the information into the computer. You must ALWAYS hit RETURN to actually enter any information into the computer. Hitting RETURN does two things: 1) it moves the cursor to a new line, and 2) it causes the computer to respond to any information you have just typed.

(NOTE: Students will forget this most frequently during the first class. When they say "nothing is happening," you say "how do you ENTER information into the computer, to get it to respond to you?")

Everybody gets a syntax or error message. Good, just what I wanted. The trouble is that you are not using the language the computer has in its ROM, which is BASIC. You may be typing in perfectly good english, but english is not BASIC, and BASIC is the language of these computers.

THE FIRST COMMAND: PRINT

BASIC is a language, just as French, Italian, and English are, with its own vocabulary, syntax, and spelling. But unlike speaking French to a Frenchman who may be able to understand you in spite of your mistakes, you cannot make a mistake on a computer and expect it to interpret what you meant.

Computers cannot think. Nor can they do anything YOU do not instruct them to do. If you do not give the computer information with the EXACT syntax, grammar, and spelling of its language, you will get error messages. So you must give meticulous attention to detail. When you get an error message and you're sure you typed it right, look again. Examine the video screen closely and you will likely find a typing error that you made.

Let's learn our first piece of BASIC vocabulary. In any programming language, instructions are called COMMANDS. And our first command in BASIC is PRINT.

(NOTE: Go to the board and write PRINT in capital letters.)

What do you think the PRINT command will instruct the computer to do? It will cause the computer to PRINT some word or phrase you also type in.

I want everyone to start on a new line by hitting RETURN, and type PRINT, then quotation marks, then whatever string of letters or words you typed earlier, then close the quotation marks. Then ENTER the command and information.

*(NOTE: At the board, you also write an example of PRINT with a string, eg:
PRINT "INTRO BASIC"
At this point many people will ask how to get rid of their previous error messages. Tell them to ignore those. While the computer is not smart, it is very forgiving and is not concerned with anything above the cursor.)*

No error messages this time. We are now using the language of the computer, with correct syntax. If you did get an error message, examine your typing carefully.

Try another PRINT command, this time with something else between the quotation marks. Also try typing PRINT

incorrectly and see what happens. Do several lines of PRINT commands. Remember to ENTER the information when you finish each command line.

WHEN TO HIT RETURN

If you have many words to enter between the quotation marks, do not hit RETURN to get to the next line. The computer will SCROLL you there as you keep typing. You can have a line longer than the 40 characters possible on the video screen by letting the computer scroll. Never hit RETURN unless you have completely finished one line and are ready to enter another command. From now on, when I refer to "a line," I mean a single command line where one command is on a line that was begun by hitting RETURN.

(NOTE: Make sure they do not put a command on the second half of a line that has scrolled. If someone says they have "run out of room" on the monitor, briefly explain that anything they type in will eventually scroll over the top of screen. When the entire class comes upon a problem or question, then it is worth stopping everything else to explain. Some errors or questions will be particularly useful to you in illustrating some point. But if only one or two people are stumped, and it is not a "useful" error, then do not demand everyone's attention or take up much class time. This calls for constant value judgments on your part.)

Now everyone has typed in at least two or three of these PRINT commands. Every command must be the first thing on a new line, at the leftmost side. How can you always get to a new line? By hitting RETURN. Then you must type the command correctly, of course. In this case we are using the PRINT command.

PRINT WITH A STRING

Look at the elements of each line. When you want the computer to PRINT a word or phrase, use the PRINT command with a STRING. A string is not a command. A string is ANYTHING surrounded by quotation marks. So my STRING on the board is INTRO BASIC. You cannot get the computer to print any words unless you put them into a STRING. So this is the PRINT command with a STRING. You are allowed to put anything you choose into a STRING--except another set of quotation marks.

Let's examine the syntax more closely: The computer also reads from left to right. When you hit RETURN, it reads the command PRINT, which is good BASIC, then reads the first set of quotation marks. These quotation marks signal the computer to accept anything it finds next as simply shapes in space, with no meaning, until it reads the second

set of quotation marks. Then the computer executes the PRINT command it read, and prints out on the screen those meaningless shapes in space (in this case English) which were read between the quotation marks.

If you make a mistake in a line and you see it before you hit RETURN, you can back up to correct it. On your computers, start typing a line and make a purposeful mistake but don't hit RETURN. Your cursor tells you you haven't hit RETURN yet because it is at the end of your string, not on the next line. Now start backing up (DEL key on the PET, DELETE BACK S on Atari, arrow left on Apple and TRS-80). As you back over the letters they disappear (except on Apple) and they are deleted (on the Apple too). When you have reached your error, start typing forward again, and enter the line as usual.

Every new line you type into the computer, after hitting RETURN, must always have a COMMAND in BASIC. A string is not a command. A string simply holds the english, gibberish, or numbers which the computer will PRINT out.

(NOTE: When a student tries to type in a string without a COMMAND, point out that he has no COMMAND on the line.)

DIRECT MODE

What happens when you hit the RETURN key? The command is carried out, the PRINT command is EXECUTED. As soon as you hit RETURN the command is executed directly, without delay. When you give the computer commands which are carried out as soon as you hit RETURN, you are instructing the computer in DIRECT MODE: You give the computer a command and the computer executes it directly.

Questions?

DIRECT MODE is fine in some cases, necessary in others, but usually you want to give the computer a longer series of commands which will be retained on the RAM. You want the computer to WAIT until you TELL it to execute the series of commands altogether. Then instead of DIRECT MODE you will use PROGRAM MODE.

PROGRAM: LINE NUMBER & COMMAND

Now pull out your sheet called "Number Guessing Game." That is a PROGRAM. What do you see is the major difference between my PRINT command and string on the board and any of the PRINT commands in this program? The number preceding each command, the LINE NUMBER. A line number plus a command makes a program.

(NOTE: Go to the board and write a 10 in front of PRINT "INTRO BASIC" .)

Everybody start on a new line and type a 10, then PRINT, then a string, and hit RETURN. Don't use an L if you mean one, or an O if you mean zero. Each is a separate character to the computer. If you are a touch-typist, you will have trouble with this at first.

What happened this time when you entered the information? Nothing. That LINE NUMBER signals the computer that you are now in PROGRAM MODE. So what is it waiting for? Another line in your program. Give it another line number, like 20, another command, another string, enter it. Keep going like this until everybody has a few lines typed in.

Remember the line number must be the very first thing after you hit RETURN. The COMMAND must be the next thing you type. One COMMAND per line number. One line number for each new line.

You have just written your first PROGRAM in BASIC. Before you were typing in isolated COMMANDS which were executed directly, in DIRECT MODE. Now the line numbers make all the difference. A program can be a single line if it has a line number. ANY program written in any programming language will ALWAYS have line numbers down the leftmost side of the screen.

(NOTE: This refers to higher level languages, not machine code, but that is material for Computer Comfort.)

RUN, OUTPUT, & LIST

Let's say you have finished putting all the commands into your program and want to see the program executed, what happens when it RUNS. To RUN a program, you use a DIRECT COMMAND -- a line number would signal the computer to wait for yet another line number and command. Looking at your "Introduction to BASIC" sheet, do you see a COMMAND that might execute, or RUN a program? RUN. Everybody get on a new line,enter RUN as a DIRECT COMMAND, NO line number this time, and watch the screen.

What you see on the monitor now is the OUTPUT of your program. When you type RUN the computer begins carrying out those COMMANDS you typed in, it executes your program. The result is the OUTPUT. The program and the output are two separate things. If you want to see the program again, to see a LIST of the commands you typed in, there is a direct command you can use. On your sheet do you see a command that might produce a LIST of your program? LIST. Both LIST and RUN are ALWAYS DIRECT COMMANDS. On a new line everybody enter LIST. Now what do you see..? The

PROGRAM.

(NOTE: Push the point that the LINE NUMBERS and the COMMANDS together make up the PROGRAM and it is the PROGRAM they see after LIST. Help them to articulate this using their new vocabulary.)

The program and the output look fairly similar at this point. But as you learn more commands, the more different a program will look from its output.

To illustrate the difference between a program and its output, think of making a cake. A computer program is a set of instructions like a recipe. If you want to produce a cake, you follow the recipe, execute those instructions, and the result is the cake. But the cake and the recipe are two SEPARATE things. The cake, the output, is the result of the recipe, the instructions. Similarly, when you enter the RUN command the computer executes your program and produces an output. To see the instructions on the computer, a list of the program, you enter..? LIST. To see the result of your program, to see it run, you enter..? RUN. Make up sentences like these using the commands and it will help you to remember what they do.

Questions?

The program you just wrote is still on the RAM, as long as you don't turn off the machine or start a new program. You can RUN and LIST that same program ad infinitum. RUN and LIST your progarm a few times again.

What about those of you who got error messages when you entered RUN? The computer cannot continue to RUN a program beyond an error. The computer may not tell you exactly what the error is, but it will at least tell you WHERE the error is, in which line number.

EDITING

Type LIST and examine the problem line.

(NOTE: Write a statement with an error at the board, e.g.:)

```
10 PRINT "FAN"
20 PRINTT "BASIC"
```

To correct the mistake, I start on a new line, type the same line number and enter the rest of the line as it should be, free of errors. When I type a new line 20, the computer AUTOMATICALLY replaces my old line 20 with the most recent line 20.

(at the board add a new line 20:)

```
10 PRINT "FAN"
20 PRINTT "BASIC"
20 PRINT "BASIC"
```

How can you be sure the computer put the new line in..? By typing LIST.

(at the board erase the first line 20:)

```
10 PRINT "FAN"
20 PRINT "BASIC"
```

Everybody correct or change a line in your program now, by typing a line number existing already in your program, followed by a command.

Make sure the exchange has taken place, by typing..? LIST. If you sit there and type in 50 different line 20s, which one do you think the computer will keep? When the computer finds two of the same line number, it keeps only the most recently typed one in the program.

You don't have to wait for an error message to correct a mistake. You can change or correct a line that you have already entered at any time by starting on a new line and retyping the line number followed by a command.

If you get an error message when you try to run a program, type LIST and look closely at the line number indicated in the error message. From now on, the first thing you should ALWAYS do when you get an error message after typing RUN, is type a LIST and examine the line where the computer found a mistake.

(NOTE: Always get them to do this before they call you over for help. You can of course teach editing with the cursor instead of retyping the line, but microcomputers are so different (particularly the Apple II and PLUS, and TRS-80 16K) in their cursor movement that it could take much time or be a mess. And they will forget 90% of the time to hit RETURN after editing with the cursor. Retyping a line by its line number emphasizes the importance of line numbers early on. Many will pick up cursor movement on their own or ask you, and you can explain it to them individually.)

(NOTE: If you are running quite behind time or you don't feel the class is ready, then save the next three paragraphs on editing techniques, or the ones on PRINT with a value, for an appropriate time.)

There are other handy characteristics about these line numbers. How does the computer know what instruction to execute first after you enter RUN? What if I type in my program out of order, say line 20, line 10, line 40, line

30..?

(NOTE: At the board, write lines out of order.)

Programs are ALWAYS executed in NUMERIC ORDER regardless of what order you type them into the computer. Whenever you type LIST, the program is listed in numeric order. Whenever you type RUN, the program is run in numeric order.

If I decide later to add a line between two existing ones, I can simply pick a number between the two existing ones, and type that number followed by a command. This is why you do not number your lines 1, 2, 3, etc. What if you decided later that you needed a line inserted between 2 and 3. You would have no whole numbers left between any existing line numbers, and line numbers must be whole numbers. Everybody now insert a line into their program between two others. It is always best to pick a number half way between any two, so you allow room for possible later additions.

Type LIST to see a neat list of your program. You can delete any line by typing the line number, and hitting just RETURN. Everybody delete a line now. Delete another. Only one line is deleted at a time. Type LIST to see the newest version of your program. Type RUN and look at the difference in the output. The computer executes only those commands in your program.

PRINT WITH A VALUE

With all this editing under your belt, let's learn the second type of PRINT statement: PRINT with a VALUE. So far you have been using PRINT with a string. What's a string..? Anything between quotation marks. PRINT can also be used with a VALUE, like this:

(at the board write:)

10 PRINT 4+7

What's different from the other PRINT statements..? No quotation marks, no string. What do you think the output will be when I type RUN..? Eleven.

When the computer begins running this program, it finds the smallest line number, 10, and reads PRINT. If next it finds no string, it had better find an expression that has some VALUE. So for output, the computer will PRINT the VALUE of that expression, which is eleven.

Everybody add a similar line to their program and RUN the program.

What happened if you typed:

(at the board write the following errors:)

```
10 PRINT 7+5=
```

You got an error message. PRINT statement syntax does not permit an equal sign when printing a VALUE.

If you typed:

```
10 8-9
```

you also got an error message. LIST will show a line numbered 108 in your program. There is no BASIC command on that line.

You must use the PRINT command to have the program SHOW the answer in the output. And you are allowed only operation symbols in the expression.

The sign for division is the backslash: / and to multiply use the asterisk: * . You can use parentheses in BASIC the same way they are used in mathematics.

(NOTE: If some one gets a math answer he didn't expect, you can explain orders of operation: with no parentheses, all multiplication and division are done first, in left to right order; then addition and subtraction are done, in left to right order. You must specify all operations, thus you cannot put a number next to a parenthesis for multiplication -- you must include the asterisk. This information may not be necessary for the whole group; some people will feel quite uncomfortable if you dwell on too much math.)

Tell me what the output will look like for this program:

(at the board write:)

```
10 PRINT "4+7="
20 PRINT 4+7
RUN
```

```
Output..?
4+7=
11
```

Remember, anything can go inside of a string. The computer does not even try to read a string, it simply prints whatever it finds between the quotation marks. What about the output for this program:

(at the board write:)

```
10 PRINT "4+7="
20 PRINT 5*3
RUN

Output..?
4+7=
15
```

No error messages. The BASIC is flawless. But the logic has gone awry. In any program the BASIC must be correct. BEYOND that, the program must be LOGICAL, make sense to a human being, to be useful. The computer only knows BASIC. The programmer must know BASIC and logic.

These examples of PRINT with a VALUE would be easier to calculate on a pocket calculator. When we get to VARIABLES, however, you will see how this type of PRINT statement can be a powerful programming tool.

Questions?

STARTING A NEW PROGRAM: NEW

Let's put a different program into the computer. The computer can hold on its RAM only one program at a time. So you must get rid of the current one to put in a new one. If you simply began typing in a new program, the computer would consider the new lines additions and/or replacements for old ones. You might end up with a very odd "cut and pasted" new program. What is one way then to clear the RAM..? By turning off the machine. But there is another way which is easier on the computer, therefore preferable.

Look on your "Introduction to BASIC" sheet. If you want to start a NEW program, what might you type..? NEW. Like RUN and LIST, NEW is always a DIRECT COMMAND. Everybody enter NEW. How can you be sure the program is gone..? Enter RUN or LIST, and nothing comes up on the screen.

THE GOTO COMMAND

Now everybody's RAM is cleared. Let's learn a new command to put in our programs. First enter a one line program: line number, plus command, plus string.

(At the board write:)

```
10 PRINT "I AM FAN"
```

On my line 20, I'm going to add a new command:

(at the board, you now have:)

```
10 PRINT "I AM FAN"
20 GOTO 10
```

What do you think the output for this program will look like? Everybody add a line 20 just like this one. Don't cast away everything else you just learned: on a new line, type 20, then the new command GOTO as ONE word, followed by the number 10.

(NOTE: Some people will also put PRINT on their line 20 with GOTO. Point out that GOTO is a separate command, therefore gets its own line and line number.)

If you make a mistake, back up and correct it or hit RETURN and start over with the line number. Then RUN the program.

What you are seeing on the screen is an infinite loop. A string is printed at the bottom of the screen as one scrolls over the top, very quickly. This will continue infinitely.

THE BREAK KEY

While a program is running, you cannot RUN or LIST or change any lines. The computer can only do one thing at a time, so if it is running a program, you must first stop the run to use another command. To break out of a running program you hit whatever keys apply to the computers you are using: RUN-STOP and RETURN on the PET, CONTROL-C on the Apple, BREAK on the Atari, BREAK on the TRS-80, etc.

Once you have broken out, the program is still on the RAM. You did not wipe it out, you just stopped it from running. So you can still LIST it, or RUN it again and break out.

Let's walk through this program as the computer would.

(NOTE: Using the program at the board, write some the initial lines of output as they would occur while you talk through the program.)

As soon as I enter RUN the computer looks for the smallest line number in my program, which in this case is 10. Remember, programs are always executed in numeric order. Line 10 says to PRINT the string "I AM FAN" . So the first visible output on the screen is:

(at the board write:)

I AM FAN

Now what comes after 10 in this program..? Line 20. Line 20 says GOTO 10, which sends the computer back to read line 10. Line 10 says to PRINT "I AM FAN" and the output looks like:

I AM FAN
I AM FAN

What comes after 10? 20--and 20 says GOTO 10, and 10 produces another I AM FAN . Now the output is:

I AM FAN
I AM FAN
I AM FAN
And 20 comes after 10, and on and on, ad infinitum. This program creates an infinite loop. It will never stop unless you break out of it.

So the syntax of a GOTO statement is: a line number, the command GOTO as one word, followed by a line number that exists SOMEWHERE in your program. GOTO is a directional command. You can send the computer anywhere in a program with GOTO, up above or down below. You can get an infinite loop only when you send the computer ABOVE the GOTO command. The Number Guessing Game program is in an infinite loop, though that output will look quite different from yours.

Questions?

*(NOTE: About now some one will ask "What's the use of this?" Tell them to look again at the number Guessing Game program, at all the lines with GOTO. Sometimes you want the computer to GOTO some other part of the program based on conditions or equations. They must learn how any command works, its capability and syntax, before they can employ it to perform useful operations. Tell them they should be getting a feel for programming: a language is made up of VERY simple commands. Each command does essentially one thing. They become useful only when a programmer can make them work TOGETHER, when the programmer uses LOGIC to build the individual commands into a program.)

You have learned an enormous amount today. You've learned the parts of the computer and how to turn it on. You've learned program commands like PRINT with a string, PRINT with a value, and GOTO. You know how to write a program, how to break out of one; how to RUN and LIST a program; how to start a NEW program. You know how to edit while on a line, or by changing a line, and how to delete a line. Spend 15 minutes before the next class reviewing so this will come back more easily when we begin class.

LESSON TWO

(NOTE: As people come in at an uneven rate, encourage everyone to go ahead and turn on the computers. Give them a few minutes practice time. This will remind them that they do know something, and help clarify what they don't understand or remember.)

REVIEW

Let's have an oral quiz/review. What do you do at the end of every line to enter information into the computer..? Hit RETURN.

What's a STRING..? Anything between quotation marks. What can go into a string..? Anything except another set of quotation marks.

What command do I use to have the computer do calculations..? PRINT with a value.

What's the difference between a DIRECT COMMAND and a PROGRAM COMMAND..? A direct command has NO line number. A program command MUST have a line number. What are some commands which are DIRECT COMMANDS ONLY..? RUN, LIST, NEW.

If I write a program and want to see the output, what do I type..? RUN. If the program has scrolled off the video screen, what can I type to see it again..? LIST.

If I want to delete an entire program, what do I do..? Type NEW, or turn off the computer, preferably the latter.

If I have typed in a line incorrectly, but don't see it until three lines later, what can I do to correct it..? Type the line over again, starting with the line number. How do I delete a line..? Type the line number and hit RETURN.

How do you break out of a program that is in an infinite loop..? (BREAK, RUN-STOP, CTRL-C, etc.)

Questions? *(Expect many.)*

VARIABLES

Today you will see why the computer is more powerful than a pocket calculator. You will learn to use VARIABLES.

Does everybody remember this from math:

(at the board you write:)

A=8

You can assign number values to letters and put the letters into equations like this:

(at the board write:)

A+1=9

(NOTE: Some people may not recall this at all but ask if this makes sense at least.)

In BASIC you also have variables, letters assigned a number value, like this:

(at the board write:)

```
LET A = 8
```

It reads like this: Let A GET eight. Use "get" because in BASIC you also write:

(at the board add another line:)

```
LET A = 8
LET A = A+1
```

Now what's A..? Nine. Try to think of the equal sign as GET: Let the variable A GET the value of itself plus one.

(at the board add another line:)

```
LET A = 8
LET A = A+1
LET A = A-5
```

Now A is..? Four. The variable starts out with the value of eight. Then A gets the value of itself, eight, plus one, so A becomes nine. Finally A gets the value of itself, nine, minus five. So after the third line the value of A is four.

The form is different from math. Remember the syntax: LET, a single variable name, equal sign, then whatever value the variable gets.

PRINT A VARIABLE'S VALUE

If I want the computer to PRINT the VALUE of A, what command would I use..? PRINT A. No quotation marks. If I write PRINT "A", then I get just an A for output. What would the output for this program look like:

(at the board write:)

```
10 LET A = 8
20 LET A = A+1
30 PRINT A
RUN
```

Output..?
9

LET does not produce any visible output, just as GOTO does not produce any visible output. If you want any printed output you must use the PRINT command.

I can have many variables in one program if I choose:

(at the board write:)

```
5 LET B = 15
10 LET A = 8
20 LET A = A+B
30 PRINT A
```

The output will be:
23

So I can use one variable to define another, but I must be sure that EVERYTHING on the right side of the equal sign has a defined value. In otherwords, you cannot have line 20 without line 5; how would the computer know what the value of B is, to add it to A?

Everybody type in a new program. Type NEW if you need to clear the RAM. Put in several LET statements. Don't forget a few PRINT statements so your output will show something.

REMEMBER this syntax: a line number, the command LET, next is a single variable name, then the equal sign, then the value.

Questions?

Here's what is going on in the computer's memory, step by step, with a LET statement. Look at the program on the board. When I type RUN, the computer finds the smallest line number which is 10 here. As soon as the computer reads LET B, it LITERALLY sets up a box or a bin, in memory, called B. Then it reads to the right of the equal sign to see what B GETS. It finds 15, and puts a 15 right

into that varialble box called B. You need to remember this tedious, step by step process to understand the way variables work and the rules for syntax.

(NOTE: At the board next to the program, you should draw a box and label it B. When the computer "reads to the right of the equal sign to see what B gets," you should draw a fifteen in the box. Likewise below, do the same with the LET A statement: draw a box labeled A, see what value A gets, and draw the eight in the box. When the value of A changes, erase the old value and draw the new one in. Always draw these labeled boxes on the board and show the value changing.)

In line 10 the computer sets up a variable box in memory called A, and puts eight in it. In line 20, the computer reads LET A, finds it already has a box called A set up; so it goes on and calculates the value of A, eight, plus the value of B, 15, and puts the new value, 23, into the A box. A variable can hold only one value at a time. Whenever the computer reads a LET statement, it will put a NEW value into the variable box, and eject the OLD value. The value can be very large, but only one value goes into a box at a time.

Because these bins are set up in memory, their values are accessible to the computer at any time WHILE THE PROGRAM IS RUNNING. So at line 30, when the computer reads PRINT A, it knows the value of A is 23 and can print 23. If I add a line 40 and 50:

(at the board write:)

```
5  LET B =15
10 LET A =8
20 LET A =A+B
30 PRINT A
40 PRINT B
50 PRINT A*B
```

the output will be..?

```
23
15
345
```

Why will you get 345 in the last line of output..? The new value of A was calculated at line 20, so that A became 23. at lines 30 and 50 then, the value of A is still 23. Line 50 does not change the value for either A or B, it simply PRINTS the product of the two values.

COUNTING TO INFINITY

As a class, let's write a program on the board that will produce this output:

1
2
3
4
5 --on to infinity, until you push (RUN-STOP, BREAK, CTRL-C, etc.). This is a progam that will count to infinity.

(NOTE: Get the class to prompt you while you write the program at the board. Put up the initial incorrect solutions they may suggest, and walk through those until they hit upon the correct solution.)

(at the board write:)

10 LET X =1 Looks pretty good. What next?
20 LET X =2 You can't type in numbers from one to infinity, so we need to increase the value of X each time. Let's change line 20:

20 LET X = X+1 Now what is the command to create an infinite loop..? GOTO. And where should the computer go?

30 GOTO 10

(NOTE: Now run this as if you were the computer, outloud at the board. Draw a box labeled X, put one in it. Then read line 20, erase the old value of X and put two in X. GOTO 10, and put one in X.)

We have a problem in our loop. Let's change line 30:

(at the board change line 30:)

```
10 LET X = 1
20 LET X = X+1
30 GOTO 20
```

(NOTE: Walk through the program again, outloud, changing the values of X, but after 3 or 4 values stop.)

But what about the output? The program is running just fine, changing values in X, but we aren't seeing anything on the output. We need a PRINT statement where?

(at the board add line 15:)

```
10 LET X = 1
15 PRINT X
20 LET X = X+1
30 GOTO 20
```

(NOTE: Run the program again, showing the single value of one for output.)

So we must also alter line 30:

(at the board change line 30:)

```
10 LET X = 1
15 PRINT X
20 LET X = X+1
30 GOTO 15
```

(NOTE: Now have everyone enter this program and see it run.)

OUT OF MEMORY (OPTIONAL)

Actually, the computer won't generate numbers to infinity because it has a fixed memory size and cannot generate numbers past a certain point. The biggest number it knows is not infinity. But at the rate of this program, which adds only one each time, you could sit here for a few days until the computer runs out of memory. You can alter the program like this:

(at the board write:)

```
10 LET X =5      ----let X equal some number under ten
15 PRINT X
20 LET X = X * X
30 GOTO 15
```

This will increase the numbers very quickly, and within just a few lines the computer will reach a point where it cannot generate a bigger number. It will say "OVERFLOW ERROR" which means you have overflowed its memory. Try it and see.

*(NOTE: Explain when they have done this that a number with an E plus a number is scientific notation: the number with the decimal point is multiplied by ten with an exponent of whatever follows the plus sign:
1.23456789E + 16 = 1.23456789 times ten to the sixteenth power.)*

INVOLVING THE USER: INPUT

When you do word processing on these computers, you are USING some one else's program. You interact with the program a great deal, answering questions, giving input, typing in your text to manipulate it, etc. In this class you have been programming, writing programs. You are all programmers. But you have also been using your programs: when you type RUN and see the output, you are the user. There has not been much distinction so far because the user has not had anything to do.

Now we are going to make a quantum leap in our programming. We are going to involve the USER. We will write a flexible program so that the user can make decisions about the output. We will allow the user some INPUT into the program. The command is INPUT. Take the counting program and let's modify line 10:

(at the board write:)

```
10 INPUT X
15 PRINT X
20 LET X = X*X
30 GOTO 15
```

The INPUT command does everything the LET command does, and a little bit more. In this program, after you type RUN, the computer reads INPUT X and sets up a box in memory called X. Then it causes a QUESTION MARK prompt to appear in the OUTPUT. Don't confuse this question mark with the one you get with error messages. This question mark is a PROMPT to the USER, the computer is prompting the user for some INPUT. And the computer won't go any further until the user gives it some information. The PROGRAMMER does not decide what the value for X will be; the USER decides. When the user finally does type a number, the program still does not continue until the user enters the number by hitting RETURN. Then whatever number the user gives is immediately sucked into that X box as the value for X.

Everybody enter this modified counting program with the INPUT command. Do not pick a value for the input until you have typed RUN and the computer prompts you with a question mark. Remember that a variable name MUST follow the INPUT command. The value for that variable goes in when the user is prompted by a question mark. After the program has run once, RUN it again and give a different number when you are prompted for some input.

Each time you run the program you, as user, can put in any number you choose. As programmer, you designed a FLEXIBLE program by using INPUT. This allows any user running this same program to pick any number. REMEMBER: Using LET statements, the PROGRAMMER decides the value for variables; INPUT statements allow the USER to decide the value.

When does the user give the computer the information..? When the question mark prompt appears, after the program has begun to RUN. You cannot give the computer information until you are asked to give the information. The INPUT command causes the computer to ask for information, and the INPUT command is only executed after you type RUN.

Questions?

DIRECTIONS WITH INPUT

If some one off the street walked in and ran your program, he would not have any idea what the computer wanted when this question mark appeared out of nowhere. So put some directions in the program that will tell the user what to do. Where in the output should the directions appear, what do you need to tell the user, and what command will you use to do it?

(NOTE: You could teach the use of INPUT as a PRINT command, e.g.: 10 INPUT "GIVE ME A NUMBER";X. However, this can be confusing when first learning INPUT, especially since it involves punctuation. In a short course it is more important that the students understand how to put commands together to make a program.)

The directions should appear first in the output, so we should add a line 5. The directions should say at least "GIVE ME A NUMBER" . How do we get the computer to print words..? PRINT with a string. Now the program looks like:

(at the board write the modification:)

```
5  PRINT "GIVE ME A NUMBER AND HIT RETURN"
10 INPUT X
15 PRINT X
20 LET X =X*X
30 GOTO 15
```

Output will be..?

```
GIVE ME A NUMBER AND HIT RETURN
?6                    --Let's say the user picks 6
6
36
1296
1679616
2.82110991E+12
7.95866111E+24
OVERFLOW ERROR IN 20
```

From now on, your most important strings will be those that contain directions or information for the user.

(NOTE: At some point, have students RUN the program and purposely enter LIST, or NEW at the question mark prompt. They will get a REDO FROM START or REENTER message. When the computer is asking for a number, the user cannot give it a word. Explain also that when a program is running, you can only run with it, or break out of it. A computer can do only one thing at a time: the programmer cannot LIST or erase a program that is in the process of running.)

RULES FOR THE NAMES OF NUMERIC VARIABLES

You do not have to pick single letters for variable names. Normally you want to use DESCRIPTIVE names, that refer to what thing that particular box is holding a value for. This is for the benefit of ONLY the programmer. The user never sees the variable names. The user gives values to the computer when prompted, but never sees where in memory the values are held.

Some computers have greater flexibility when it comes to naming variables, but these rules should always work for any microcomputer:

(NOTE: Either write this at the board or make a handout similar to the one in the appendix of this booklet:)

Variable Names:

1. Must begin with a letter.
2. Can be alpha-numeric (letters and numbers).
3. Must be less than six characters long.
4. Two names in the same program must be different in first two characters.
5. Cannot contain words reserved by the computer, like RUN, LIST, etc.

(NOTE: Put up a few examples on the board as you read through the rules, to illustrate legal and illegal names. Reserve words are different for some micros, so it is best to avoid any commands from BASIC, even within a name. LETTER, for instance is an illegal name on the Apple II and PLUS. Those computers will find LET within the word. LIST will show: LET TER . The spacing will tip you off about reserve words, if a manual is not available.)

Questions?

While the name can be descriptive, the description does not determine what goes into the box. How the programmer manipulates the variables determines what they hold.

Pull out the sheet with the Big Bucks Banking program on it. Looking through it, tell me what the names of the variables are..? BAL and AMT.

(NOTE: Some one will say AMOUNT and BALANCE. Ask them look again, what are those variables really named? If the class has much trouble understanding the difference between the value and the variable name, keep explaining. If they don't have trouble, then move on to typing it in.)

Now, what do you think BAL will hold..? The balance. And what will AMT hold..? The amount of a check or deposit. I gave the variables descriptive names, which I MADE UP, so that I, as programmer, could pick them out easily and

remember what value I wanted them to represent. I could call them CAT and DOG and the program would run just as well, with no visible difference to the user.

I want you to think of these boxes as physical places, that hold number values. But don't confuse the variable NAME with the VALUE. My house is a place that holds me. The name of the place is 1000 Fulton Street. But I am not 1000 Fulton Street, I am what goes into 1000 Fulton Street.

BIG BUCKS BANKING PROGRAM

Let's walk through this program before you start typing it. Essentially this program could be used to figure the balance in your check book.

First the user is given directions to enter a beginning balance. Where does that value go in memory..? Into BAL. More directions follow: enter checks as negative numbers, type negative one when you wish to stop. Then the user is prompted for an amount of a check or deposit. That value goes into AMT.

Line 90 has a new command, IF-THEN, but it works very much like the english equivalent. If the user types in a negative one, what is supposed to happen according to the directions..? The program should end. So IF AMT is equal to negative one, THEN GOTO 130. After line 130, the program ends.

At line 100, BAL is added to AMT to get the new BAL, thereby updating the balance everytime line 100 is executed. Line 120 prints out what the new balance is. That semicolon allows the computer to PRINT both the string and the value of BAL on the same line of output.

At line 125 the computer goes back to line 70 to ask for another check or deposit.

For now, go ahead and type in the Big Bucks program and RUN it. Remember: When you are programming, you do not put a value in for the INPUT statements. Who gets to put the value in..? The user. You copy this program like a XEROX machine, paying careful attention to details. When you RUN it, then as user, answer the questions it asks you.

(NOTE: Give students a chance to get the program in, correct it, and possibly run it a few times. Point out when they run the program the second time, that all the previous values are gone. Very slow typists may not get a chance to run the program. Have them run some one else's before they begin the IF-THEN command. Ask everyone to keep the program on the RAM for later modification.)

PROGRAMMING: SYNTAX VS. THEORY

There is nothing wrong with the BASIC anywhere in the banking program. But THEORETICALLY, there are a few flawed parts of this program. Look at those directions on line 60: "when you wish to stop type negative one." What is wrong with that in the abstract sense? What if some one wants to enter a check for one dollar? A negative one would end the program. They could write a check for $0.9999999 cents, or for $1.001, but not for just one dollar. That is one limitation.

(NOTE: Students may suggest changing the flag to zero, as a more logical solution at this point, since checks are not written for zero dollars.)

Now why would Wells Fargo certainly not want to buy this program from me? Will banks let you take out money infinitely? This program will. There is nothing in here to stop you from having a negative balance. This program will let you withdraw into deep oblivion, and continue to calmly report "NEW BALANCE IS $" no matter how negative your balance is. This program is not written to distinguish a negative balance from a positive one. In a few minutes, however, you will modify the program to change that.

THE IF-THEN COMMAND

Let's examine line 90, the IF-THEN command. What happened when you entered a negative one for the amount..? The screen said "END OF TRANSACTIONS" . What happened when you entered anything else for the amount? The program went on and gave the new balance. When the AMT was NOT equal to negative one, the computer did NOT GOTO line 130. It went ahead to line 100 and calculated the new balance.

(At the board write:)

IF (a condition) THEN (do something)

IF-THEN statements work very much the same way in both english and BASIC. IF a condition is satisfied, THEN do something. When the condition is NOT satisfied, the computer does not read to THEN, but continues on to the next line number and command.

THEN should always be followed by a command in BASIC. Conditions in BASIC are expressed with mathematical symbols:

(at the board write:)

IF (a condition) THEN (command)

=	is equal to
<	is less than
>	is greater than
<=	is less than or equal to
>=	is greater than or equal to
<>	is not equal to

(NOTE: Explain the meaning of each symbol, as you write the symbol at the board. You might add an equation as another possible condition, like: IF A+B=C THEN (command).)

You can pose a condition between two variables or between a variable and a number. In BASIC when a condition is NOT satisfied, the computer drops to the next line number and command, ignoring whatever might follow the THEN. At line 80, if you entered 50.00, is the condition at line 90 satisfied..? No. AMT is not equal to negative one. The computer drops to line 100. If you enter negative one at line 80 the condition at line 90 IS satisfied, and the computer executes the command which follows THEN: GOTO 130.

(NOTE: If more illustration is needed, pull out the Number Guessing Game program and have them examine those condition statements, plugging in imaginary numbers for both GES and NUM.)

The syntax of an IF-THEN command is: IF followed by a condition expressed with a mathematical symbol, THEN follows the condition, and a command in BASIC follows THEN.

MODIFYING THE BANKING PROGRAM WITH IF-THEN

Now I want you to modify the banking program you have on the RAM, using an IF-THEN command. I want you to insert the necessary line or lines, so that when the user is overdrawn the program tells the user "YOU ARE OVERDRAWN" . That is the only requirement. What happens next is up to you. You can have the program end; or let the user continue making transactions; or you can get fancy and alter the program so that the user can only make deposits until he is back in the black.

You will need to decide: what is the condition to test for; what should happen when the condition is satisfied; where should the condition statement be within the program. In any serious programming, most of your time is spent with pencil and paper, not at the computer. Planning is crucial. So work out a solution on paper first. Then type it into your program. Be sure to try several different inputs to test your solution. Your program may work with one input but not all inputs. You want a solution that works for ALL situations.

Keep track of all your solutions, especially the ones that don't work. We will talk about all of them to see what went wrong and what went right. There are an infinite number of workable solutions to this problem. Some will be better or "cleaner" than others, and some will be equally good.

(NOTE: Give them about 15 minutes, 20 if possible, to work on this. If someone gets a solution that works, tell them to work out still another solution.)

Let's start with the solutions that did not work. You give me the line number and commands that you added, plus any changes you made in other lines.

(NOTE: Solicit solutions from the class. Put the added line and any changed lines up on the board. Draw the variable boxes at the board and update their values as you walk through the program. The unworkable solutions shown below are some that come up most frequently.)

```
85 IF BAL < 0 THEN PRINT "YOU ARE OVERDRAWN"
```

You enter RUN, and the computer first asks you for the beginning balance and sets up a box called BAL at line 20. Let's start with a balance of $50, so a 50.00 goes into the BAL box. Then we get some more directions, we are asked for the amount of a check or deposit, and the computer sets up a box called AMT.

Let's overdraw immediately with a check for $100.00. So negative 100.00 goes into AMT. At line 85, is the value in BAL less than zero, is 50 less than zero? No. The condition is NOT satisfied, the computer drops down to the next line number. At line 100 our new balance is calculated to be -50.00. We are overdrawn, but the computer never said so.

Let's continue back to line 70, and deposit $100 this time, to get back in the black. AMT gets 100 now. At line 85, is BAL < 0? Is negative 50 less than zero? Yes. The condition is satisfied, so the computer reads past THEN, and prints "you are overdrawn."

In fact, however, line 100 claculates our new balance to be -50 plus 100, which equals positive $50.00. We need to test the balance immediately after the new balance is calculated, at line 100. The condition is not in the right place. Let's move the new line to 105:

```
105 IF BAL <0 THEN PRINT "YOU ARE OVERDRAWN"
```

(NOTE: The whole program now looks like:)

```
10 PRINT "GIVE ME YOUR BEGINNING BALANCE, NO $ SIGNS."
20 INPUT BAL
30 PRINT
40 PRINT "GIVE ME THE AMOUNT FOR EITHER A CHECK OR"
50 PRINT "A DEPOSIT. WRITE CHECKS AS NEGATIVE NUMBERS."
60 PRINT "WHEN YOU WISH TO STOP, TYPE -1."
70 PRINT "AMOUNT PLEASE $" ;
80 INPUT AMT
90 IF AMT = -1 THEN GOTO 130
100 LET BAL = BAL + AMT
105 IF BAL < 0 THEN PRINT "YOU ARE OVERDRAWN"
110 PRINT
120 PRINT "NEW BALANCE IS $" ;  BAL
125 GOTO 70
130 PRINT "END OF TRANSACTIONS."
```

Let's RUN again. We will deposit $100.00 for the initial balance, so 100.00 goes into BAL. For our first transaction, let's write a check for $100.00, so -100.00 goes into AMT.

Line 100 calculates our new balance to be zero, and BAL gets 0. Is the condition at line 105 satisfied, is BAL less than zero? No. The computer drops down, and continues the program. So at least it doesn't tell us we are overdrawn when in fact we aren't. But let's continue back to line 80 and overdraw this time. Let's write a check for $50.00, so -50.00 goes into AMT.

Line 100 calculates the new balance to be negative 50. At line 105, is the condition satisfied, is BAL, -50, less than zero? Yes, the condition is satisfied and the computer prints "you are overdrawn." The program continues on, still letting the user overdraw forever. But the criterion I set forth has been met. The program tells the user when the user is overdrawn.

To reach this solution at line 105, the first thing you had to decide was what condition to test for: when is the user overdrawn? Most of you decided the user was over drawn if the balance was less than zero. That translates right into BASIC: IF BAL < 0. Then what should happen when the condition is satisfied? The computer should print "you are overdrawn." So: IF BAL <0 THEN PRINT "YOU ARE OVERDRAWN." That all translates quite smoothly into BASIC. Last, you had to decide where to put the test. After some experimentation, it became clear that the balance should be tested AFTER the new balance is calculated. With this solution, the program tells the user that he is overdrawn; then the user is told the new balance, and the program continues.

Where else could you have this same test? At line 123. This solution would produce an output that tells the user a new balance, and then tells the user that he is overdrawn. So the order of information to the user is reversed. This

solution also allows the user to continue withdrawing ad infinitum.

*(NOTE: Time permitting, you can work out the solution for having the program end after it tells the user he is overdrawn, e.g.:

```
10 PRINT "GIVE ME YOUR BEGINNING BALANCE, NO $ SIGNS."
20 INPUT BAL
30 PRINT
40 PRINT "GIVE ME THE AMOUNT FOR EITHER A CHECK OR"
50 PRINT "A DEPOSIT. WRITE CHECKS AS NEGATIVE NUMBERS."
60 PRINT "WHEN YOU WISH TO STOP, TYPE -1."
70 PRINT "AMOUNT PLEASE $" ;
80 INPUT AMT
90 IF AMT = -1 THEN GOTO 130
100 LET BAL = BAL + AMT
105 IF BAL <0 THEN GOTO 127
110 PRINT
120 PRINT "NEW BALANCE IS $" ;  BAL
125 GOTO 70
127 PRINT "YOU ARE OVERDRAWN"
130 PRINT "END OF TRANSACTIONS."
```

The new lines are 105, 127, and 130.)*

Another solution?

```
85  IF AMT > BAL THEN PRINT "YOU ARE OVERDRAWN"
```

Begin the program with RUN. Let's start with a deposit of $50.00. So the INPUT command puts 50.00 into BAL. After we are asked for the amount of a check or deposit, the computer sets up a box called AMT. Let's overdraw by entering a check for $100. So -100.00 is put into AMT at line 80.

At line 85, is the value in AMT greater than the value in BAL, is -100 greater than 50? No, the condition is not satisfied, the computer drops straight to the next line number, 90, and continues running the program. Even though we are overdrawn the computer does not tell us so.

With our new balance at -50.00 after line 100, let's say we deposit $100.00 when asked for the next amount, to get back in the black. So AMT becomes 100.00. At line 85 again, is AMT greater than BAL, is 100 greater than -50? Yes, the condition is satisfied, the computer reads past THEN and PRINTS "you are overdrawn." But in fact the new balance, calculated at line 100, is positive $50.00.

(NOTE: At this point or earlier, students may suggest changing line 85 to AMT < BAL. Try their initial remedies, but eventually prompt them, lead them, to the correct solution.)

The relationship between the amount and the balance is a little trickier. Some people decided that if the balance is less than zero, the user is overdrawn. They set up a relationship between just the balance and zero. Here you are trying to establish a relationship between the amount and the balance, that will indicate when the user is overdrawn.

Essentially the program must determine when the user is trying to write a check for more money than is in the balance. The relationship between the amount and the balance is this: If the amount of a check (a negative number) is greater than the balance, then the user is overdrawn. The directions indicate that checks must be written as negative numbers, so that line 100 will reflect the true balance. Thus the relationship is set up as:

```
85 IF -AMT > BAL THEN PRINT "YOU ARE OVERDRAWN"
```

(NOTE: The whole program will look like this:)

```
10 PRINT "GIVE ME YOUR BEGINNING BALANCE, NO $ SIGNS."
20 INPUT BAL
30 PRINT
40 PRINT "GIVE ME THE AMOUNT FOR EITHER A CHECK OR"
50 PRINT "A DEPOSIT. WRITE CHECKS AS NEGATIVE NUMBERS."
60 PRINT "WHEN YOU WISH TO STOP, TYPE -1."
70 PRINT "AMOUNT PLEASE $" ;
80 INPUT AMT
85 IF -AMT > BAL THEN PRINT"YOU ARE OVERDRAWN."
90 IF AMT = -1 THEN GOTO 130
100 LET BAL = BAL + AMT
110 PRINT
120 PRINT "NEW BALANCE IS $" ;  BAL
125 GOTO 70
130 PRINT "END OF TRANSACTIONS."
```

Let's walk through this version. We will start with a balance of $50, so 50.00 goes into BAL at line 20. Let's immediately overdraw by writing a check for $100.00. AMT gets -100.00 in it.

At line 85, is the negative of negative 100 greater than 50? A negative negative makes a positive, so is positive 100 greater than 50? Yes, the condition is satisfied, the computer reads past THEN and PRINTS "you are overdrawn" and indeed we are. The new balance is calculated at line 100, and line 120 PRINTS the new balance as -$50.00.

We go back to 70. This time let's deposit $100 to get back in the black, so 100.00 goes into AMT. At line 85, is negative AMT greater than BAL? Is negative 100 greater than negative 50? No, the condition is not satisfied, and

the computer drops down to line 90. And indeed, our new balance is positive 50, as line 120 will tell us.

The visible output for this program is the same as the solution we used at line 105. The user is told that he is overdrawn, then told the new balance, and the program continues. Two entirely different tests of relationships produced identical outputs.

(NOTE: Again, time permitting, you can rework this solution slightly to have the program end if the user overdraws.)

SAVING AND LOADING (OPTIONAL)

(NOTE: If tapes and diskettes are available to the students, you can have them SAVE their programs for more modifications. In the next class, they can LOAD the banking program. A handout on how to SAVE and LOAD for each type of computer in the room is advisable.)

SUMMARY

Today you learned how to write a flexible program using numeric variables and the INPUT command. You also learned how to set up IF-THEN commands to test for certain conditions involving variables. Next time we will look at string variables; and how to create a loop which stops at a given point instead of going on endlessly.

(NOTE: Modifications for the banking program are good homework assignments. The variations are endless. When you give them new specifications for the program, tell them to use this program as reference, but to write their own from scratch. Modifying some one else's program is actually more difficult than writing one from scratch. An over-modified program usually becomes awkward and hopelessly convoluted. At this stage, flowcharting is not essential. Have them write their ideas down on paper, in english, as clearly as possible. Then translation into BASIC should be smooth.)

LESSON THREE

STRING VARIABLES

We modified the Big Bucks Banking program so that it is slightly more sophisticated. It will recognize a negative balance. However, we still have the one dollar check problem. The program will end if the user writes a check for one dollar.

We should fix the banking program so that it will ask the user, after each transaction, whether he wishes to continue. Rather than entering negative one, the user can respond with a yes or no to continue or stop the program.

What sort of input is the response going to be? A word, not a number.

Thus far we have been using only numeric variables. In BASIC you CANNOT put words into numeric variable boxes. If you wanted to PRINT the word NO, what would you have to make NO into? A string. What we need then is a variable box to hold strings.

BASIC allows string variables as well as numeric variables, but you CANNOT mix them. Fortunately, however, the rules and theory work the same way for both string and numeric variables. The only difference is that you must signal the computer, let it know that you are setting up a box to hold strings rather than numbers.

RULES FOR STRING VARIABLE NAMES: THE DOLLAR SIGN

Look again at your sheet with the rules for numeric variable names. ALL of those rules apply to string variable names, plus one more: the last character of every string variable name MUST be a DOLLAR SIGN, not sitting off one more space to the right, but PART OF THE NAME. So we can have a string variable called A$, or B$.

(NOTE: At the board write A$ and B$, and pronounce them as "A string and B string.")

It helps to read that dollar sign as the word "string," for example: A string, or B string.

The dollar sign has nothing to do with money or numbers. It is simply the symbol used to signal the computer to set up a variable box to hold strings. Think of it as a fancy S, which stands for "string."

You can use PRINT, INPUT, and LET commands with string variables, and you can set up conditions to compare string variables as well. You can write a program:

(at the board write:)

```
10 LET PHRS$ = " IS A NICE NAME."
20 PRINT "WHAT IS YOUR NAME"
30 INPUT NAME$
40 PRINT NAME$ + PHRS$
```

Remember to put those quotation marks around the words when using the LET command. Otherwise, it is not a string. And of course, you must use a legal string variable name, which includes the dollar sign.

Again I have made up descriptive variable names, just as BAL and AMT are somewhat descriptive names. PHRS$ is short for phrase, and NAME$ is more obvious. But just because I have called a variable NAME$ does not mean the computer knows it will hold a name. The computer only knows that NAME$ will hold a string. And what can go into a string..? Anything. The user can type in any jibberish he chooses and the computer will not generate an error message.

*(NOTE: If there is an Atari in the room, you must also explain the DIM command for Atari string variables. The Atari is more particular about the useage of string variables, so the program must include information about the POSSIBLE MAXIMUM size of the string variable box before anything is put into it. Putting the DIM statements at the very beginning of the program is a good habit. Each string variable must have a DIM statement. For the above program, on the Atari, you would add:

```
5 DIM PHRS$(16), NAME$(30)
```

It is best to always DIM string variables which get their value through INPUT for the MOST room they might need. Not too many people will have names longer than 30 characters. Remember, too, that spaces count as characters within a string.)*

What will the output for this program look like..?

(at the board write:)

```
WHAT IS YOUR NAME
? FAN    ----(I will use my own name)
FAN IS A NICE NAME.
```

Fortunately the user does not have to know about using the

quotation marks. When the computer reads INPUT with a string variable, it will accept whatever the user types in as a string.

Look closely at the string in line 10: " IS A NICE NAME." How would the output look if I changed that string and left out the space preceding IS:

(at the board change line 10 to:)

```
10 LET PHRS$ = "IS A NICE NAME."
20 PRINT "WHAT IS YOUR NAME"
30 INPUT NAME$
40 PRINT NAME$ + PHRS$
```

The output would be:

(at the board write:)

```
WHAT IS YOUR NAME
?  FAN
FANIS A NICE NAME.
```

The computer does not print any spaces unless you put them into the string. If you want a space to appear between words, you must include the space within the string. Let's put the space back into the string at line 10.

(Erase the change and put the space back in at the board.)

Using the original version of the program, let's run it again with a different INPUT:

(at the board write:)

```
RUN
WHAT IS YOUR NAME
? 4+8
```

What do you think will be the next line of output..? ERROR REDO FROM START..? No. Remember, what can go into a string..? ANYTHING.

Maybe the output will be:
12 IS A NICE NAME..? No. A PRINT command used with a string or a string variable will not calculate the value just because the string happens to contain numbers. The actual output will look like:

(at the board write:)

```
RUN
WHAT IS YOUR NAME
? 4+8
4+8 IS A NICE NAME.
```

Since strings can hold anything -- letters, numbers, symbols -- string variables can hold anything. The characters within the string are shapes in space to the computer. The computer will not recognize numbers within a string or within a string variable.

MODIFYING THE BANKING PROGRAM WITH STRING VARIABLES

Let's rewrite the banking program so that it asks the user before each transaction: "DO YOU WISH TO CONTINUE, YES OR NO?" Of course, we will need to modify the instructions, and add another INPUT statement with a variable to hold the response. Then we will need a line that sets up a condition so that if the response is "no," the program will go to the end. We will delete line 90, since the program should no longer go to line 130 if the user writes a check for one dollar.

(NOTE: If time allows, let the students work on this by themselves for a while.)

Those qualifications translate easily into BASIC. We will modify line 60 to say:

(write lines 60, 63, and 67 at the board:)

```
60 PRINT "DO YOU WISH TO CONTINUE, YES OR NO"
```

Now we need to get the response from the user..?

```
63 INPUT RESP$
```

Then we should test the response to see if it is "no."

```
67 IF RESP$ = "NO" THEN GOTO 130
```

Don't forget the quotation marks around the string. Think about this test. What if the user responds "yes"..? Then the condition is not satisfied, and the program continues.

COORDINATING THE WHOLE PROGRAM

When you set up a condition it must be coordinated with the whole program. We could have set up line 67 to test for a "yes" response instead of a "no" response. In that case, if the condition were satisfied, the program would have to go somewhere else rather than line 130.

Here, however, we have set up our condition at line 67 to test for a "no" response. There is still one more alteration needed for our banking program to make it work as we intended, with these new lines on the board, 60, 63, and 67. Let's suppose the user responds "yes" at line 63. The condition is not satisfied, the program continues. The

new balance is calculated, perhaps a line 105 will be there to test for an overdrawn balance. Then what happens? The computer goes to line 70 and asks the user for an amount. It did not ask the user if he wishes to continue or not. So line 125 should be rewritten:

(at the board write:)

125 GOTO 60

Now the user is asked after each transaction if he wants to continue. And he can write a check for one dollar without ending the program.

In final form the program will look like:

```
10 PRINT "GIVE ME YOUR BEGINNING BALANCE, NO $ SIGNS."
20 INPUT BAL
30 PRINT
40 PRINT "GIVE ME THE AMOUNT FOR EITHER A CHECK OR"
50 PRINT "A DEPOSIT. WRITE CHECKS AS NEGATIVE NUMBERS."
60 PRINT "DO YOU WISH TO CONTINUE, YES OR NO"
63 INPUT RESP$
67 IF RESP$ = "NO" THEN GOTO 130
70 PRINT "AMOUNT PLEASE $" ;
80 INPUT AMT
100 LET BAL = BAL + AMT
105 IF BAL < 0 THEN PRINT "YOU ARE OVERDRAWN"
110 PRINT
120 PRINT "NEW BALANCE IS $" ;  BAL
125 GOTO 60
130 PRINT "END OF TRANSACTIONS."
```

But what if, when asked if he wishes to continue, the user responds "no way!" indicating he emphatically does not want to continue..? Is "no way!" equal to, the SAME AS, "no"? It is not. With "no way!" as the response, the condition is not satisfied and the program continues. You may have to put in more explicit instructions for the user, such as "type only yes or no." Or you may put in more tests. Or you can leave the program as is and rely on users to follow the given directions.

WHEN THE USER DOES NOT FOLLOW DIRECTIONS (OPTIONAL)

It is sometimes a good idea, sometimes necessary, to write a program which can recognize mistakes by the user, and then remind the user how to do it correctly. I could put a test into this program that said if the user's response is not equal to "no" or "yes" then print "you must answer no or yes," and then return the user to the response input.

When you add parts to a program to take into account mistakes by the user, that is called "idiot proofing." Programmers have an adage that says if you completely idiot-proof a program, only a complete idiot will use it. That's a bit strong. But at some point, the programmer must count on the user to do what the program asks the user to do.

But what is that point? When a knowledgeable programmer uses a program, error messages do not appear cryptic to him, nor do they alarm him. But if you buy your computer solely to do word processing, you may not know, or care to know, anything about programming.

With the advent of microcomputers, and with manufacturers trying to get them into everybody's house, to reach a non-programming market, there has been a growing emphasis on "user-friendly" programs.

"User-friendly" programs are programs which have a LOT of idiot-proofing built into them. Such a program does not come to a dead stop when the user makes a mistake. The error messages are in clear english, not numeric jargon, like the ones in Atari BASIC. "User friendly" programs are easily intelligible to non-programmers, and have commands logical to non-programmers.

For commercial software programs to succeed, the art of "idiot-proofing," or of making a program "user-friendly," must grow into a science.

THE FOR-NEXT COMMAND: A CONTROLLED LOOP

We are going to learn the FOR-NEXT command and then apply it to the banking program. The FOR-NEXT command allows you to create a controlled loop, to repeat a task a FIXED number of times, rather than repeating it infinitely as with GOTO.

You have a sheet with FOR-NEXT and STEP on it. I will put that first program on the board:

```
10 FOR X = 1 TO 5
20 PRINT X
30 NEXT X
40 PRINT "NOW X IS"; X
50 PRINT "SO COMPUTER DROPPED OUT OF LOOP."
```

The output for the program is shown on the sheet, too:

```
1
2
3
4
5
NOW X IS 6
SO COMPUTER DROPPED OUT OF LOOP.
```

Lines 10 through 30 are part of the FOR-NEXT LOOP. That loop is executed five times before X gets out of the range of one to five. When the value for X is six, the loop ceases.

The FOR-NEXT command allows the programmer to create a loop that is completed a fixed number of times, rather than infinitely. The loop itself will always be defined as the lines between the FOR and NEXT commands. Let's walk through this like the computer to see how it works:

(NOTE: Put an X box on the board when you read the FOR command and change the values as the computer would.)

When you enter RUN, the computer, as always, looks for the

smallest line number, which in this case is 10. Line 10 reads "for x gets one to five." The FOR command causes the computer to identify the beginning of a FOR-NEXT LOOP; and to set up a variable box in memory called X, just as the LET and INPUT commands do. Then the computer reads to the immediate right of the equal sign to see what X gets, in this case it is one. The TO part of the command describes the loop's range for X: X can have a value up to five and still continue the FOR-NEXT LOOP.

So at line 10 the upper boundary of a loop is defined by FOR; a variable box called X is set up; X gets an initial value of one, and can have values up to five for the loop.

Line 20 is PRINT with a value. Whatever the value of X is, will be printed in the output each time this line is executed.

At line 30 two things happen: First the NEXT X command causes the value of X to be bumped up one. The value of X is now..? Two. The second thing which happens is that the computer checks line 10, with the FOR command, to see if the value of X is still within the range. Is two within the range of one to five..? Yes. So the computer goes through the loop again.

On this second trip through the loop the output from line 20 will be..? Two, since the value for X is now two. At line 30, the value for X is bumped up to..? Three. Is three within the range..? Yes, the computer goes through the loop again, beginning with the first command following the FOR statement.

The value three gets printed. Then X is bumped up to four, which is within the range, so four gets printed on output. Then X is bumped up to five. Five is still within the range, so the computer goes through the loop again, and prints a five for output.

Now at line 30, NEXT X bumps the value of X to six. Is six within the range..? No. The computer does NOT go through the loop again but drops out of the loop, to the line number which follows the NEXT command.

At line 40 the output shows "now X is " and what is the value of X now..? Six. So the computer PRINTS the value of X as six, and then PRINTS "so computer dropped out of loop."

It is important to understand that the variable box X can hold ANY number, but only numbers up to five will keep the loop going.

In this FOR-NEXT loop, we have only one other program line that is executed each time through the loop. But the FOR and NEXT command lines may be separated by many lines, even

an entire program.

Be careful of the syntax. You may never use a FOR command without a NEXT command, and vice-versa. FOR-TO-NEXT might have been a better name for this command, because you must ALWAYS have TO, spelled out not as a dash, on the same line with the FOR command. FOR should always be immediately followed by a numeric variable name, then the equal sign, then TO with some values on either side of it. The NEXT command must be followed by the same numeric variable name which follows FOR.

What would my output look like if I changed line 10 to:

(at the board change line 10 to:)

```
10 FOR X = 5 TO 9
20 PRINT X
30 NEXT X
40 PRINT "NOW X IS "; X
50 PRINT "SO COMPUTER DROPPED OUT OF LOOP."
```

Output..?

```
5
6
7
8
9
NOW X IS 10
SO COMPUTER DROPPED OUT OF LOOP.
```

I make any range I choose using the FOR-NEXT command.

USING STEP (OPTIONAL)

(NOTE: STEP is not terribly useful to the students at this point and may be skipped to devote more time to in-class programming.)

Adding a fourth command to the FOR-TO-NEXT combination you can bump up the value of the loop variable by something other than one. Look at your sheet at the second example:

(at the board:)

```
10 FOR X = 1 TO 10 STEP 2
20 PRINT X
30 NEXT X
```

The STEP command affects how much the value of X is increased each time the computer reads NEXT X. So the values for X in this program will be: 1, 3, 5, 7, 9, and when X is bumped to 11, the computer drops out of the loop and this program ends. Will the output show 11..? No. When the value of X is eleven the computer will not go

through the loop, so line 20 will not be executed.

STEP can be used for moving backward through a range too. The next example on your sheet demonstrates that. When moving in negative steps, however, remember to state your range from high to low, five to one in this case. If it were written:
10 FOR X = 1 TO 5 STEP -1
the value of X would be one the first time through the loop, NEXT X would bump X to zero, which is no longer within the range, and the computer would drop out of the loop. The programmer probably did not intend for the loop to be executed just once, but no error message will appear.

NESTED LOOPS

Look now at the last program on your sheet about FOR-NEXT LOOPS. Remember that the FOR and NEXT commands can be separated by many commands. In fact one loop may contain another loop inside of it. Then they are called "nested loops":

(at the board write:)

```
10 FOR X = 1 TO 3
20 PRINT "I AM"
30   FOR Y = 1 TO 3
40   PRINT "FAN"
50   NEXT Y
60 NEXT X
```

The output is shown on your sheet. You can see that each time one "I AM" appears, three "FAN"s follow it. The Y loop is completely nested inside of the X loop. Let's walk through this as the computer would.

(NOTE: At the board draw the variable boxes as the program sets them up and change their values when the computer would.)

After you enter RUN, the computer reads line 10, sets up a box called X, puts a one in it, and notes that the loop range will be one to three. Line 20 causes I AM to appear on the output.

At line 30 the computer sets up another box, called Y this time, puts a one in it, and notes the range for the loop is one to three. Line 40 produces FAN on the output. Line 50 bumps the value of Y up to two. Remember that the computer does not try to start through a loop again until it reads the corresponding NEXT command. Is two within the range..? Yes, and the computer continues through the Y loop again.

FAN gets printed how many times on the output..? Three times. When Y is four, the computer drops out of the loop to the next line number which is 60.

Line 60 now bumps the value of X up to two. Is two within the range..? Yes. The computer then starts through the X loop again. Line 20 produces another I AM on the output.

And here we are again at line 30. The computer reads FOR Y gets one to three, and puts one in a variable box called Y. The Y loop is executed three times again, until Y becomes four. Then the computer drops to line 60. X is bumped up to three, the X loop starts again, and so on.

After the computer drops out of the Y loop this time around, line 60 bumps X up to four. Is four within the range..? No. There is no line number which follows 60, so the program ends. Remember, every time the computer reads line 30 FOR Y gets one TO three, the Y box will get a one, and the FOR-NEXT loop with Y will start over again. Coming out of the loop at line 50, Y will have a value of four. But reading line 30 causes the computer to put a one into Y and start the loop over.

On your sheet, I have drawn lines on the left side of this program to show the boundaries of each loop. Whenever you are nesting loops, or using two or more loops in one program which are not nested, these lines cannot cross over one another, they cannnot intersect. On paper, write out the program, draw these lines to mark out the loop boundaries and make sure they don't cross over one another. At line 50, the computer cannot calculate the value of NEXT X, the first loop initiated, before it drops out of the most recently initiated loop, in this case Y. After nesting its way into loops, the computer must then find each variable's NEXT command in REVERSE order. The most recently initiated loop must always be completed before the computer can execute the NEXT command for a previous variable loop.

With nested loops, the variable in the first loop initiated should correspond to the variable in the last NEXT command of the entire program; the second variable loop initiated should correspond to the second from the last NEXT command; the third variable loop initiated should correspond to the third from the last NEXT command, etc.

At line 50, I could not have NEXT X because after line 30, the Y loop becomes the most recently initiated loop. The computer must find NEXT Y before it finds NEXT X.

*(NOTE: Give an example on the board of a program with two loops that are not nested, and the output e.g.:

```
10 FOR X = 1 TO 3
20 PRINT "I AM"
30 NEXT X
40 FOR Y = 1 TO 3
50 PRINT "FAN"
60 NEXT Y
```

Draw the loop lines and show the output:

```
I AM
I AM
I AM
FAN
FAN
FAN
```

)*

Now write a program, using only three PRINT statements, that will produce this output:

(at the board write:)

```
I AM
FAN
ALLEN
ALLEN
ALLEN
FAN
ALLEN
ALLEN
ALLEN
I AM
FAN
ALLEN
ALLEN
ALLEN
FAN
ALLEN
ALLEN
ALLEN
```

*(NOTE: Give them time to work this out. As with the banking program, put up initial incorrect solutions and lead them to the correct one. At the board, you will finially have:

```
10 FOR X = 1 TO 2
20 PRINT "I AM"
30   FOR Y = 1 TO 2
40   PRINT "FAN"
50     FOR Z = 1 TO 3
60     PRINT "ALLEN"
70     NEXT Z
80   NEXT Y
90 NEXT X
```

)*

MODIFYING THE BANKING PROGRAM WITH FOR-NEXT

Take out your original version of the Big Bucks Banking program. I want you to modify that program so that the program asks the user how many transactions he wishes to make. The program then asks for each amount and calculates each new balance as many times as the user specified for the number of transactions.

If you get that one done quickly, modify the later version of the banking which also tells the user when the user is overdrawn.

(NOTE: Again give them time to work this out. You may not want to solve this as a class, but let each person work with it on his own. Have the faster students modify the program so that a user with a negative balance can only deposit until he is in the black again, AND the user gets charged a returned check fee of $10. Since this is their last class, give them as much programming time as possible. Walk around helping with the problems and posing them to quicker students. Eventually you can write a solution on the board.)

(Below lines 60 and 125 were changed; lines 63 and 65 were added; line 90 was deleted:)

```
10 PRINT "GIVE ME YOUR BEGINNING BALANCE, NO $ SIGNS."
20 INPUT BAL
30 PRINT
40 PRINT "GIVE ME THE AMOUNT FOR EITHER A CHECK OR"
50 PRINT "A DEPOSIT. WRITE CHECKS AS NEGATIVE NUMBERS."
60 PRINT "HOW MANY TRANSACTIONS DO YOU WISH TO MAKE"
63 INPUT TRANS
65 FOR X = 1 TO TRANS
70 PRINT "AMOUNT PLEASE $" ;
80 INPUT AMT
100 LET BAL = BAL + AMT
110 PRINT
120 PRINT "NEW BALANCE IS $" ;  BAL
125 NEXT X
130 PRINT "END OF TRANSACTIONS."
```

THE LAST WORD

Today you have learned to use string variables. You have also learned how to create a controlled, or limited loop using FOR-NEXT. You have modified the banking program so that it is now both more user-friendly and sophisticated.

You know that a program is a set of instructions and information. The instructions and information can be loaded in through cassette or diskette, typed in by you the programmer, or put in by the user. The computer will keep this information in its memory only as long as the machine is turned on.

By now you have gotten a sense of what programming is about. It is both simple and complex: very simple commands which are put together in a complex logic. Logic and tedious attention to detail are the primary concerns of elementary programming. As you must with any foreign language, you must memorize the vocabulary, spelling, and syntax of a programming language like BASIC. The computer can do math and understands the BASIC language, but otherwise it knows no information that you do not give it.

END OF LECTURE

APPENDIX OF HANDOUTS FOLLOWS

APPENDIX

OF

HANDOUTS

INTRODUCTION TO BASIC PROGRAMMING

BASIC: GETTING TURNED ON

BIG BUCKS BANKING, AUTOMATED CHECKING PROGRAM

RULES FOR VARIABLE NAMES

FOR-NEXT STEP

NUMBER GUESSING GAME

A WORD ON BUYING SOFTWARE

INTRODUCTION TO BASIC PROGRAMMING

Language -- BASIC

hardware

software -- ROM eg BASIC : 1000 bytes =
RAM workspace : 1K of memory

Input -- Program = a set of instructions

Output -- result of the instrucions

DIRECT COMMANDS = command e.g. PRINT "INTRO BASIC"
(NO line #)

RUN (execute program)

LIST (list the program)

NEW (start a new program)

CSAVE/SAVE
CLOAD/LOAD

PROGRAM COMMANDS: LINE # + COMMAND = A PROGRAM

10 PRINT "NOW WE HAVE A PROGRAM"

PRINT with a string e.g.: PRINT "INTRO CLASS"

PRINT with a value e.g.: PRINT 4+8

GOTO plus line #

LET plus a variable: LET A, LET A$

INPUT plus a variable: INPUT A, INPUT A$

IF (a condition) THEN (a command)

FOR--NEXT : FOR variable name = value TO value
loop
NEXT variable name

BASIC: GETTING TURNED ON

Boot (v.): to load software into hardware.
DOS = Disk Operating System

* RULE OF THUMB: If it doesn't work, try again; and again; after 3d try, ask for help

APPLE II :

1) Turn on monitor (33 UHF).

2) Turn on computer.

3) Carefully insert DOS master 3.3 into disk drive #1 and close door.

4) Hold down the CTRL key, and the B key simultaneously, and hit RETURN.

5) Type: PR #6 and hit RETURN.

6) When the red drive light goes out, type: FP and hit return.
NOTE: > indicates Integer BASIC; a bracket indicates Applesoft BASIC.

APPLE II PLUS :

1) Turn on monitor (33 UHF).

2) Shut disk drive doors.

3) Turn on computer and hit RESET immediately (or CTRL and RESET).

ATARI :

1) Turn off disk drive power.

2) Turn on monitor (2 or 3, check computer switch).

3) Turn on computer.

COMMODORE PET & CBM :

1) Turn on computer.

TRS-80 :

1) Turn on monitor.

2) Turn on computer and hit ENTER.

BIG BUCKS BANKING

AUTOMATED CHECKING ACCOUNT PROGRAM

```
10 PRINT "GIVE ME YOUR BEGINNING BALANCE, NO $ SIGNS."
20 INPUT BAL
30 PRINT
40 PRINT "GIVE ME THE AMOUNT FOR EITHER A CHECK OR"
50 PRINT "A DEPOSIT. WRITE CHECKS AS NEGATIVE NUMBERS."
60 PRINT "WHEN YOU WISH TO STOP, TYPE -1."
70 PRINT "AMOUNT PLEASE $" ;
80 INPUT AMT
90 IF AMT = -1 THEN GOTO 130
100 LET BAL = BAL + AMT
110 PRINT
120 PRINT "NEW BALANCE IS $" ;  BAL
125 GOTO 70
130 PRINT "END OF TRANSACTIONS."
```

(NOTE: The strings in PRINT statements 40-60 can be rewritten if the output is too inconveniently truncated. A more convenient flag to end the program would be 0, instead of -1. However, I want the students to think about the theory problem with a negative one, that a writing a check for one dollar will also ends the program. Once they realize this, they can change the flag.)

RULES FOR VARIABLE NAMES

NUMERIC VARIABLE NAME:

1) must begin with a letter.

2) can be alpha-numeric.

3) two numeric variables in one program must differ in first two characters of name.

4) should be less than seven characters.

5) cannot contain RESERVE WORDS.

LEGAL NUMERIC VARIABLE NAMES:	ILLEGAL NUMERIC VARIABLE NAMES:
A12	12AB
ACT3	ACCOUNT
VALUE	NUM#1.2
MONTH	ACT$
	CAT (AT is a RESERVE word in Applesoft)

STRING VARIABLE NAME:

1) the last character of the name must be a dollar sign.

2) must begin with a letter.

3) can be alpha-numeric.

4) two string variables within one program must differ in first two characters.

5) should be less than seven characters.

6) cannot contain RESERVE words.

LEGAL STRING VARIABLE NAMES:	ILLEGAL STRING VARIABLE NAMES:
NAME$	NAME $
NAME4$	YOURNAME$
MONTH$	MONTH
	LETTER (contains RESERVE word LET)

(Some microcomputers are more sensitive to RESERVE words than others. Check the machine's BASIC manual.)

FOR-NEXT STEP

A CONTROLLED LOOP (not infinite) USING FOR-NEXT AND SOMETIMES STEP

```
10 FOR X = 1 TO 5  -----(gives initial value AND range
20 PRINT  X                    for loop)
30 NEXT X          -----(bumps X up one each time
40 PRINT "NOW X IS " ; X                    it hits NEXT)
50 PRINT "SO COMPUTER DROPPED OUT OF LOOP."
```

Output looks like:

```
1
2
3
4
5
NOW X IS 6
SO COMPUTER DROPPED OUT OF LOOP.
```

To go up by more than one: -----Values for X wil be:

```
10 FOR X = 1 TO 10 STEP 2                    1
20 PRINT X                                   3
30 NEXT X                                    5
                                             7
      last time X is PRINTED on output:      9
            computer drops out of loop:      11
```

Go backwards: ------------------Values for X will be:

```
                                             5
10 FOR X = 5 TO 1 STEP -1                    4
20 PRINT X                                   3
30 NEXT X                                    2
                  last time X is PRINTED:    1
              computer drops out of loop:    0
```

STEP is used on the same line as the FOR command. If variable increases by one each time, no STEP is necessary. You can have: STEP -2.5, STEP 4, STEP 3.3, STEP -2, etc.

Nested Loops: ----------Output will be:

```
10 FOR X = 1 TO 3                  I AM
20 PRINT "I AM"                    FAN
30   FOR Y = 1 TO 3                FAN
40   PRINT "FAN"                   FAN
50   NEXT Y                        I AM
60 NEXT X                          FAN
                                   FAN
                                   FAN
                                   I AM
                                   FAN
                                   FAN
                                   FAN
```

NUMBER GUESSING GAME*

* to save time, you need to type in only those lines which have asterisks beside them.

```
10 REM GUESSING GAME
20 REM INSTRUCTIONS FOLLOW
30 PRINT "I WILL THINK OF A RANDOM NUMBER FROM 1 TO 100.
YOU TRY TO GUESS IT."
40 PRINT "I WILL TELL YOU IF YOU ARE TOO HI OR TOO LO, SO
YOU CAN GUESS AGAIN."
50 REM COMPUTER GENERATES A RANDOM NUMBER
*60 LET NUM = INT(100 * RND(1)+1)
*70 PRINT "O.K. I HAVE A NUMBER."
80 REM USER TRIES TO GUESS NUM
*90 ?:?:? "YOUR GUESS IS  " ; : INPUT GES
100 REM COMPUTER TESTS GES AGAINST NUM
*110 IF GES < NUM THEN PRINT"TOO LO.TRY AGAIN." : GOTO 90
*120 IF GES > NUM THEN PRINT"TOO HI.TRY AGAIN." : GOTO 90
*130 PRINT "THAT'S IT!!! YOU GUESSED MY NUMBER."
*140 PRINT "LET'S PLAY AGAIN."
150 FOR P = 1 TO 500 : NEXT P
*160 GOTO 60
```

NOTE: If you are using a TRS-80, line 60 will read:
*60 LET NUM = RND(100)

If you are using the PET 8K machines, lines 110-120 are too long for stacking commands. You must rewrite the program with the GOTO commands on separate lines.

A WORD ON BUYING SOFTWARE
AND
SOME RETAIL PRICES FOR SOFTWARE PACKAGES

There are three important things (among others) to consider when shopping for software: Cost, Capability, and Complexity.

The prices below are retail. They are higher than the "sale" prices, or catalog/magazine prices. And usually when you buy a complete new computer system, several packages are thrown in as part of the bargain. In short, there are many prices available below retail.

In a given category, a difference in cost usually means a difference in capability. For example, DB Master has features comparable to all those of PFS, and then some more which PFS does not have.

The key question concerning capability is can the package do the job you need it to do? PFS is one of the best packages around. But if you need to send your files to another computer via a telephone hookup (telemodem), then you must look for a different data base software package. PFS will not do it.

Complexity is the most difficult characteristic to judge. It is UNNECESSARY complexity you should watch out for. WordStar is a very complex package, and takes a few weeks to learn. But it is an extremely capable package, and probably the best word processing package currently available. Magic Window can be mastered in four hours maximum. But it does not do very much. Read magazine reviews (e.g. INFO WORLD, COMPUTE, PERSONAL COMPUTING), and ask users, to find out about a particular package.

WORD PROCESSING PACKAGES:

Magic Window	by Artsci, Inc.	$100
Screen Writer	by On Line Systems	$130
with dictionary		$200
Apple WriterII	by Apple Computer Inc.	$150
Word JugglerIII	by Quark Engineering	$295
WordStar	by MicroPro	$375
Spell Binder	by Lexisoft	$495

DATABASE MANAGEMENT:

PFS	by Software Publishing Corp.	$125
DB Master	by Stoneware	$229
VisiFile	by VisiCorp	$250

FINANCIAL PLANNING:

VisiCalc	by VisiCorp	$250